ISSN 0265-0919

90p 72

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

File Edit Search Format Font Style
Untitled
6 lines/inch
THE OMNI-READER
The OMNI-READER is a revolutionary new Text Reader or Optical Character Reader.
It costs a fraction of the price of your computer system. It reads the printed word and translates it into an electronic signal capable of being recognized and down loaded into most computers and word processors. The need for slow and tedious retyping of already printed data is eliminated.
The quick brown fox can jump from the OMNI-READER into
TEST SHEET
THE OMNI-READER
The OMNI-READER is a revolutionary new Text Reader or Optical Character Reader.
It costs a fraction of the price of your computer system. It reads the printed word and translates it into an electronic signal capable of being recognized and down loaded into most computers and word processors. The need for slow and tedious retyping of already printed data is eliminated.
The quick brown fox can jump from the OMNI-READER into
FOOTCOPY
HEMLINE
NUMERIC
TYPEFACE
OMNI-READER

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

BRIGHT SPARK We discuss the problems that need to be solved before intelligent robots can be brought to life — 1421

HARDWARE

AN EYE FOR DETAIL Your computer could soon be reading these very words with an Oberon Omni-reader — 1429

SOFTWARE

TINY TURTLE We compare the Amstrad version of Dr LOGO with other implementations now available — 1434

COMPUTER SCIENCE

AND SO FORTH Our introduction to FORTH illustrates its similarity to LOGO — both languages were designed to control devices — 1436

JARGON

FROM RANDOM ACCESS TO RASTER SCANNING A weekly glossary of computing terms — 1440

PROGRAMMING PROJECTS

ON YOUR MARKS . . . We provide the routines that get our game of Go underway — 1425

MACHINE CODE

SERIAL DRAMA A look at the facilities of the Spectrum Interface 1 serial port of use to the machine code programmer — 1438

WORKSHOP

PERFORMANCE PIECE The completed MIDI interface can now be used to play a farewell symphony — 1432

INDEX A complete index to issues 61 to 72 — INSIDE BACK COVER

Next Week

- **Like many other manufacturers, Acorn have upgraded their most popular micro to include extra features. We take a look at the much talked about BBC+.**
- **In our machine code series, we examine the LAN facilities provided by the Interface 1 for the Spectrum.**
- **We begin a new Workshop project to build a digital multimeter.**

QUIZ

1) How does the Oberon Omni-reader detect whether a line is being read from left to right, or vice versa?
2) What is happening when a MIDI timer is in free-run mode?
3) What would be the equivalent of the LOGO command FORWARD 50 in FORTH?
4) What is the major problem in generating a long series of random numbers on a computer?

Answers To Last Week's Quiz

1) The purpose of the Cray 1's front-end interface is to allow the computer to be able to read or write data to whatever mini or mainframe computer is fitted.
2) In music programming, 'step-time' means that each individual note and space is programmed separately. 'Real-time' programming is performing the piece and recording it as it is played.
3) The Magic Mouse Controller program provides the drivers that enable the on-screen sprite to be controlled from the mouse. It is not an applications program in itself, but is provided to enable the mouse to be available within the user's own programs.

Coming Up...

- Our machine code series on useful aspects of the operating systems of four popular home micros will conclude with a look at the Amstrad CPC 464/664 OS in detail.
- A programming project to design a spreadsheet in BASIC for the Spectrum, Commodore 64, BBC Micro and Amstrad CPC 464/664.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designers** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Marcus Jeffery, Martin Young, Joe Pritchard, Steve Malone, Steven Vickers, Steve Colwill, Harvey Mellar; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY CRISPIN THOMAS

BRIGHT SPARK

ROBYN BEECHE

As we have seen so far in our series on artificial intelligence, there is no shortage of theories and ideas of what robots should, and possibly will, do. But the actual problems besetting the development of intelligent robots are many and varied. We focus our attention here on the gap between theory and fact.

In no other field of computing is the gap between fact and fantasy as wide as in the area of robotics. The science-fiction robots of Hollywood are capable of walking, talking and plotting the overthrow of humanity. Up the coast from Hollywood, however, at Stanford University, the latest research robots can scarcely blunder through a crowded room without knocking over the furniture.

Of course, robots have productive roles to play in car factories and elsewhere in industry. But such systems, typically for paint-spraying or arc-

Cybernetic Starlet

Practical robots are considered to be a modern development, but Rosa Bosom (Radio-Operated Simulated Actress Battery Or Stand-by Operated Mains) pictured here with her inventor, Bruce Lacey, was built in 1966 to play the Queen of France in a production of The Three Musketeers at the Royal Court Theatre, London. Constructed from government surplus relays and motors, Rosa accepts commands in the form of musical tones, which are converted into movement, and is also equipped with ultrasonic sensors to detect and avoid obstacles. In her career as an actress, Rosa mimed her words to taped speech, while her lips were moved by remote control. More interestingly, from a cybernetics point of view, Rosa can interact with a second robot called Mate.

Rosa has appeared in several exhibitions, including Cybernetics Serendipity at the ICA, and more recently she won Andrew Logan's Alternative Miss World contest in London

welding, merely repeat sequences of pre-programmed operations without variation. They are so 'dumb' that if there is a break in the conveyor belt bringing work to them, they will blindly spray or weld the empty air.

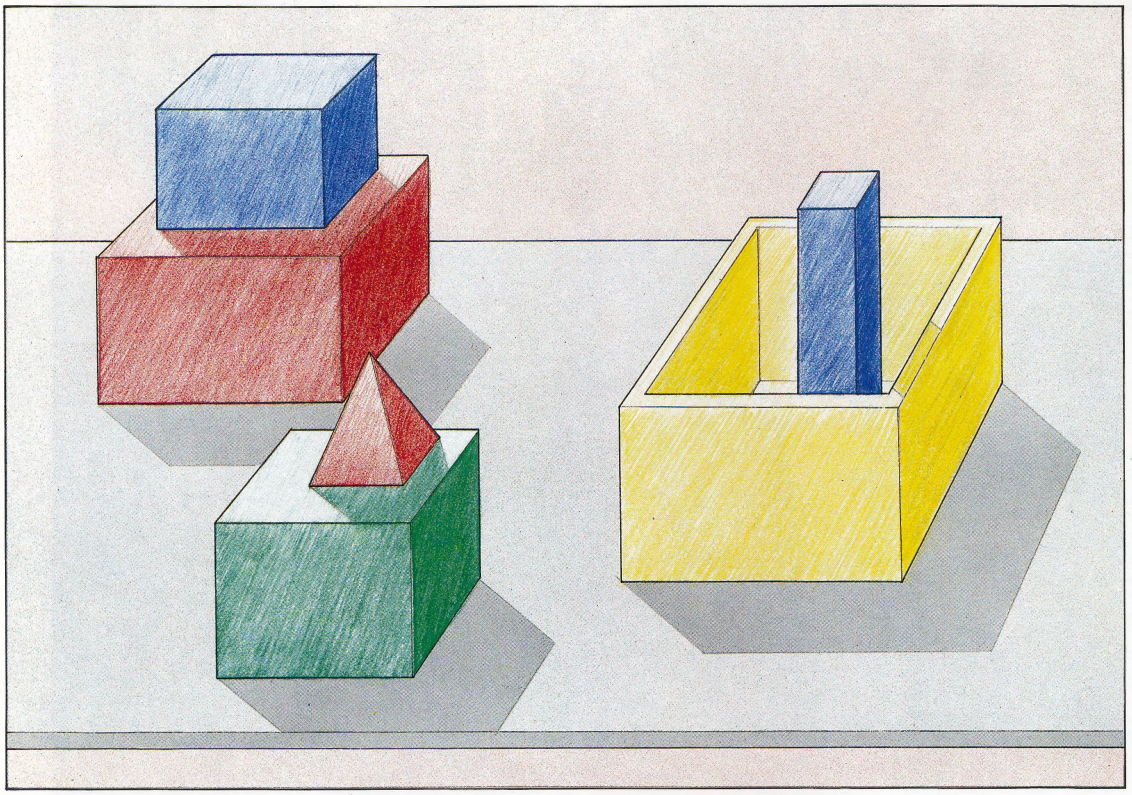
The reasons for this gaping disparity between PR and performance are complex; but they can be summed up in two words: 'perception' and 'planning'. In the first place, today's robots cannot make 'sense' of their sensors. In the second place, they would not know what to do if they could. This is why robotics is such a good testing ground for AI theories.

Robotics draws on all the facets of artificial intelligence because its aim is to produce an artefact that can cope with the real world. Therefore, a robot must make sense of its surroundings.

It is not too difficult to hitch up a computer to a variety of input devices — television cameras, heat sensors, ultrasound scanners, pressure pads, and so on — which may well give it access to information that humans cannot obtain directly, such as infra-red or ultraviolet light. But unless the robot inhabits a very tightly controlled environment, it will not understand what its sensors signify. Human perception, as we saw in the instalment dealing with computer vision (see page 1381), is an ongoing process of psychological modelling based on what is happening in the real world.

It is one thing to devise an efficient algorithm for finding a route through a maze held in a computer's memory and displayed on a screen. It is quite another matter to use that algorithm to steer a robot around an urban environment — where there may be push-chairs, discarded beer cans, bits of overgrown hedge and other unsuspected obstacles in the way.

In the real world, Murphy's Law reigns supreme. This is the reason why Micromouse contests (see page 721) are so popular. In these competitions, a computer-controlled 'mouse' has to navigate to the centre of a complex table-top maze. Methods for doing so have been known for many years, at least in theory. In practice, however, the walls are never quite straight, there are slippery patches where previous contestants have spilled oil, and so on. Under such circumstances, robustness and adaptability matter more than algorithmic elegance.



KEVIN JONES

Not The Real World
The 'Blocks World' is a highly abstracted model of the real world that enables AI scientists to evaluate different methods of scene analysis, text understanding and problem solving. However, translating successful Blocks World methods to the real world can be difficult

In most other fields of AI, the programmer can retreat to a 'microworld' of his own making, such as with a chess-playing program that operates in an abstract environment. An expert system is concerned with facts, not things. This explains why AI workers are so fond of the so-called 'Blocks World'.

THE BLOCKS WORLD

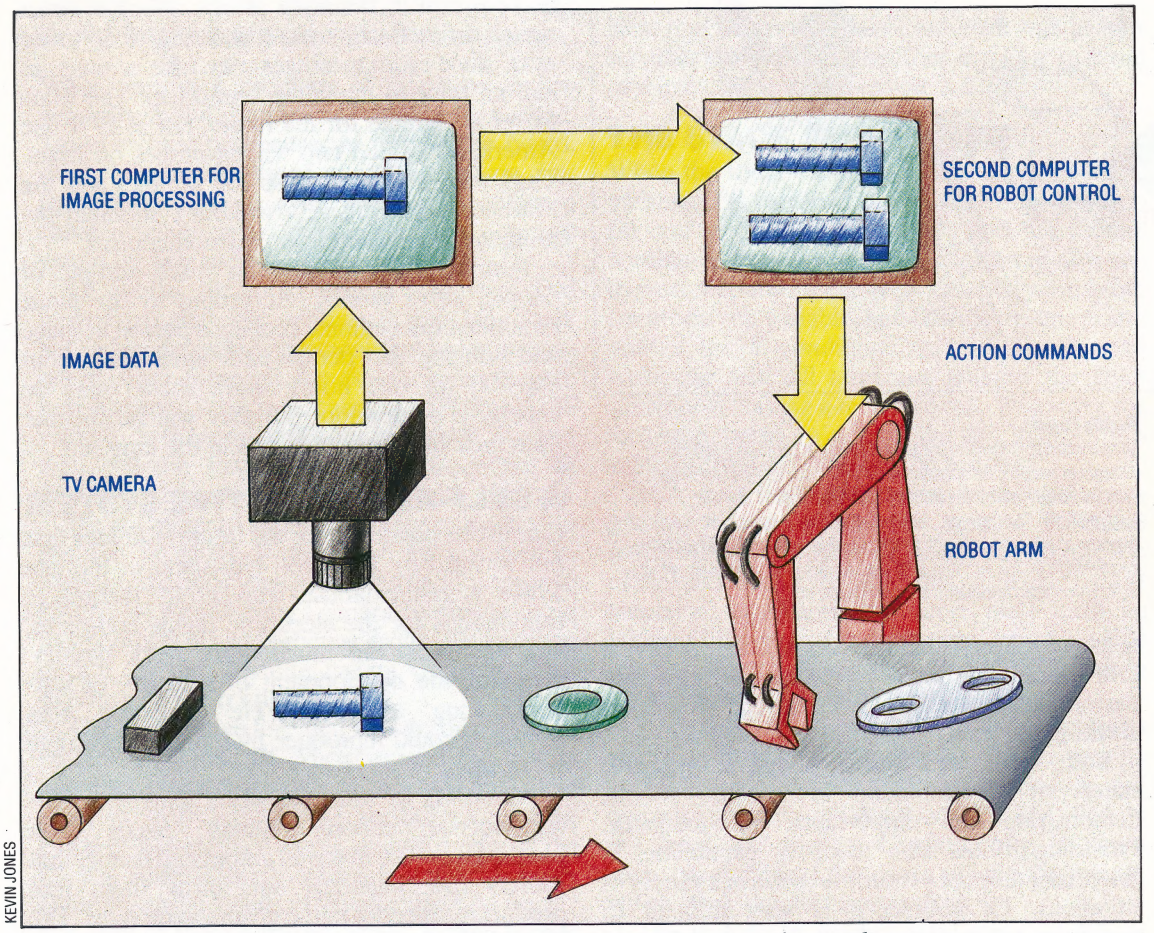
The Blocks World is a simplified schematic environment containing children's coloured building blocks. Not real building blocks, of course, but formalised *representations* of those blocks. It is a favourite haunt of AI scientists, whose programs can solve problems and manipulate objects within it without leaving the comfort of their CPUs.

The roboticist has no such refuge. Robots must face the vicissitudes of reality as best they can. An intelligent robot — one that can move around under its own initiative — must be able to build a

cognitive model of its environment. In addition (to live up to decades of advance publicity) it ought to be able to respond to spoken commands. And since all its responses cannot be thought out in advance, it would be highly desirable for it to have some degree of learning ability. That is why there are really no *intelligent* robots. Before one can be built, these AI problems will have to be solved:

- Computer vision.
- Speech understanding.
- Problem-solving in a dynamic environment.
- Machine learning.

and that is a minimal list.



KEVIN JONES

Conveying The Idea
A seeing robot may be used to recognise and pick out certain items from a conveyor belt. In this case the robot may need two computers — one to interpret incoming data from the television camera, and another to control the grab mechanism

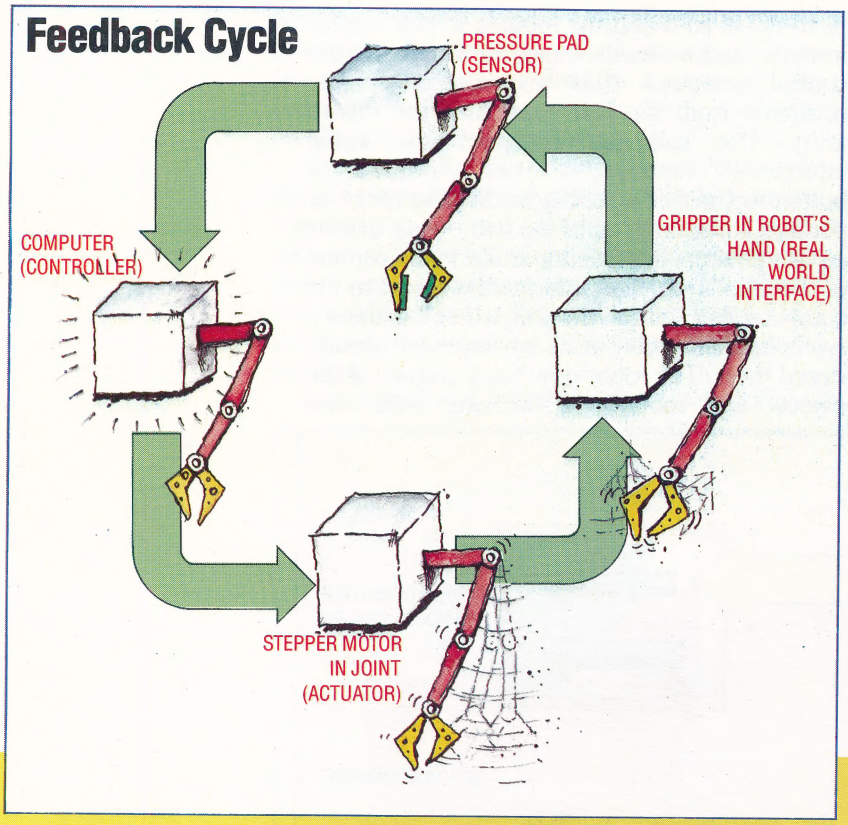
Modern industrial robots tend to be designed for highly structured environments. Typically, a robot is nothing more than an arm-like device under computer control, integrated into an automated factory or workbench. Its 'hand' can have tools bolted on and a human operator can 'teach' it a series of movements that it can repeat whenever needed. It can memorise but not modify such action sequences. Very often it operates in a setting that is so tightly regulated that it needs no perceptual abilities at all.

Some robots, however, have taken a small step into the real world. These robots are a little less dependent on people or other machines providing them their work. They can, to some degree, go looking for it.

The most common way of giving sense to a robot is through vision. A video camera takes pictures (of an object the robot is to pick up, for instance) and passes the image information to the control computer (this may be a separate computer from the one controlling the robot's arm). The computer processes the visual information supplied — using top-down or bottom-up methods (see pages 1381 and 1401), or perhaps a combination of the two — and detects certain patterns that are important in the context of the robot's job. This information is sent to the computer that controls the arm, where it updates a symbolic description of its environment already stored there. The robot now has a picture of the objects and tools it is working with, and instructions on what to do.

The great advantage of even limited robot vision is that the robot can be fed parts in various orientations, and possibly differently shaped parts as well. It can recognise the type of part it is dealing with and the angle at which it has arrived, so it can handle it in the appropriate way. A completely 'blind' robot would try to pick up each item in the same manner, probably with disastrous results.

Another way of making robots more 'sensitive' is through touch. Engineers can attach force-sensors or pressure-pads to the end of the robot's hand. When these come in contact with some object, information is relayed back to the controlling computer. If the robot bangs against a wall, the force-sensor will send back information so that it can modify its arm trajectory to avoid repeating the mishap. The standard promotional



MIKE CLOWES

Nimble Fingers
To manipulate objects accurately a sense of touch is needed. Whenever we pick up objects we estimate the weight of the object in order to determine the strength of grip required to lift it successfully. The process, which has been modelled with varying degrees of success in robot applications is essentially a feedback cycle where tactile sensors feed data back to the controller, which in turn tightens or loosens the grip on the object

display for touch-sensitive robots is to have it first drill a hole in a piece of milled steel and then perform the same trick using an egg — without cracking the shell.

Both vision and touch, though at primitive states of development in current robots, demonstrate the importance of feedback. Providing robots with feedback capabilities is essential if they are to become useful in a variety of situations. Of course, it is very difficult to transform information about the outside world into a form that the robot's computer can use to guide its actions. One problem, among many, is transforming the data in 'real-time' (fast enough for the robot to respond to what is happening before the perceived event has ended). The small minority of robots that have been equipped with sensory devices and some ability to interpret the feedback are known as 'second generation' robots.

In the instalment dealing with natural language (see page 1361), we saw that continuous speech understanding was the hardest of the four types of language-processing problems. The main difficulty is that the robot has no control over what is being said to it.

Nevertheless, it is possible to give a robot a useful degree of speech recognition (as opposed to speech understanding). This just means that the robot can be trained to respond to a small number of words and phrases in the appropriate way. It has less understanding of what is being said than a dog that can 'sit', 'heel' or 'fetch'; but in a factory where the human workers may have their hands busy, even such simple linguistic abilities can prove worthwhile.

Some of the most interesting developments in robotics have arisen from space exploration. NASA engineers have designed a Mars Rover, a robotic vehicle that can trundle around the surface of the planet Mars. The vehicle will have sensors, such as television cameras, to pick up information from its surroundings and enough intelligence to make use of it. It simply cannot afford to await signals from Earth before deciding what to do next, since radio messages may take as much as half an hour to make the round trip. The Mars Rover is planned for the future, but in 1976 the Americans landed two Viking probes on Mars. They were able to analyse soil samples and the Martian atmosphere without moment-by-moment supervision from 'mission control'.

Putting robots to work in space, instead of people, saves money. According to the latest American calculations, keeping a person in space and bringing that person back alive costs about $10,000 for one hour! Space robots may be expensive to develop, but they will be cheaper than their human equivalents in the long run.

A SELF-REPRODUCTIVE SYSTEM

One exotic possibility for the future is an idea first put forward by John von Neumann in the late 1940s. A mathematician who laid the theoretical foundations for the electronic digital computer, von Neumann did much other work besides, including the development of a theory of self-reproducing automata. He proposed that machines could reproduce by following a simple set of rules.

According to von Neumann's scheme, a self-reproductive robotic system needs four components. The first is an automated factory that gathers raw materials and turns them into products according to the given instructions. The second component is a duplicator that copies these instructions. The third is a controller that passes the instructions to the duplicator for copying. The fourth and final part is the set of instructions themselves, which tells the system how to construct a complete new automatic factory out of the products it has made.

These were just theoretical concepts for almost 40 years, until NASA researchers proposed an engineering blueprint for a self-replicating system on the moon. Such a space factory would contain a 'universal constructor' that would take parts fabricated by the production unit and build a new factory with them — including, of course, another universal constructor. It would make use of the raw materials on the moon and would not require servicing from Earth. It could also provide itself with a steady supply of new robot workers.

ON YOUR MARKS...

Having outlined the general rules of Go and provided listings for the game's I/O routines, we now need only add a few remaining miscellaneous routines that will allow us to play the game proper. These include the stack data structure and the useful implementation of recursion.

When writing a program of this type, a few generally useful routines will usually be needed. These include procedures to update the board, check for the legality of moves and so on. For instance, in a chess-playing program, a routine to check whether or not the king is being attacked would be very useful.

Of course, we could write a routine to find moves that place the opponent's king into check and, when defending, positions that keep our king out of check. However, it makes far more sense to have a general routine that assigns a TRUE or FALSE value to 'is the king of COLour in check?'. We can then call the routine with different values of COLour, and then decide what to do with the result.

We are going to adopt a similar approach with our Go program. Rather than having specific routines to decide what to do with various types of groups, we'll implement a general routine to count the members of any particular group. We can then decide, in the separate calling routines, what significance the results have.

The main procedure that will do this is called PROCsearch in the BBC Micro version. It is passed two parameters: the position of any stone in the group we wish to count (P%); and the colour of the group of stones (C%). The procedure then adds 1 to CSTN%, which is used to total the number of stones in the group, and looks at the four adjacent stones. For each of these adjacent stones, it then carries out the same procedure, but only if the stone hasn't already been counted. Of course, we already have a procedure to carry out this search in four directions from a given stone, so the obvious way to count the adjacent stones is for the procedure to call itself! This is an example of recursion, and may seem rather complicated at first. To make things a little clearer, here is a structured breakdown of the search routine:

```
SEARCH (from-position, for-colour)
  IF from-position is on the board
    AND from-position contains a stone of for-colour
    AND we have not already counted from-position
  THEN
    mark from-position, so we don't recount it
    add one to CSTN% (our count variable)
    SEARCH (north of from-position, for-colour)
    SEARCH (east of from-position, for-colour)
    SEARCH (south of from-position, for-colour)
    SEARCH (west of from-position, for-colour)
  ENDIF
END SEARCH
```

Note that the recursive calls are dealt with as four separate statements, rather then in a loop to avoid problems when the recursion is deep.

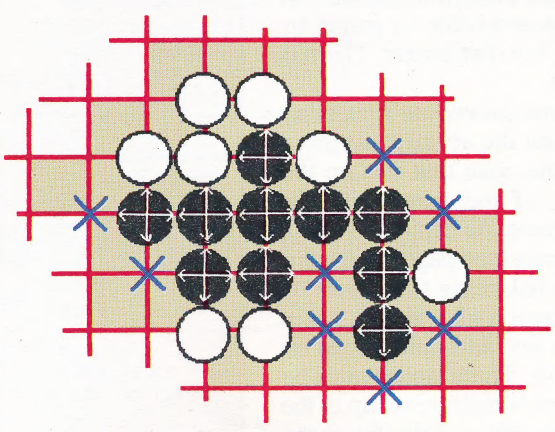
THE STACK STRUCTURE

When the routine calls itself, the 'from-position' parameter is replaced by the new from-position (in one of the four directions) and the routine continues. The new from-position may also call the search routine recursively, creating yet another from-position variable, and so on. At some point, one of the IF conditions will be false, so the particular call will terminate without the THEN statements making a recursive call. Whenever the recursion 'backs up' a level like this, the previous value of from-position will be restored.

This 'stack' data structure can be thought of as a pile of plates. When the routine is called recursively, a new plate will be added onto the top of the stack. However, when this particular level of recursion terminates, this same plate must be taken off the stack (and the same variable value will be restored). This is known as a LIFO (Last In First Out, see page 948) structure.

Unfortunately, though BBC BASIC supports recursion through PROCedure parameter passing and localised variables, the Commodore 64, Sinclair Spectrum and Amstrad CPC 464/664 do not have this facility. Consequently, in these routines, you'll find the arrays called SK%() or s(), which stack the necessary values.

This is done by holding a pointer (STACK% or



Counting Heads

The search routine, which is used to evaluate the status of groups on the board, incorporates (or simulates) recursive procedures to search in all four directions from every stone in the group. The routine terminates when a liberty or an enemy stone is reached

equivalent), which keeps a record of the topmost position of the stack. Every time something is placed on the stack, this pointer is incremented, and whenever something is removed, the pointer is decremented. The appropriate variable is then reset with the value at the top of the stack.

A by-product of this search routine is that the recursion always terminates at the edges of the group it is looking at. When we reach these edges, we can check whether they are at the edge of the board or contain a stone of the opposite colour. If neither of these conditions is true, then it must be a liberty of the group. Consequently, the second part of this routine counts the number of liberties associated with the group, using the variable CLIB%. Again, these positions must be marked once they have been counted, or it would be possible to count them twice.

When counting a group of stones, we will always call the routine PROCcount, which in turn calls PROCsearch. This will initialise the count variables, and clear the markers after using the search routine.

When marking positions, you'll notice that we use the actual board itself, to avoid double-counting. (This is illustrated on page 1395.) Having marked the board, however, we have to be able to remove the markers before continuing. Again, we've used a general routine, PROCclear, which logically ANDs the appropriate board bytes with 3, which removes the marker in bit 2. Since the colours are contained in the least significant two bits of each board byte, and the stone and liberty markers are held in the third and fourth bits, the call PROCclear(3) is ideal. For example:

Byte:	A	B	C	D	E	F	G	H
Mask:	0	0	0	0	0	0	1	1
Result:	0	0	0	0	0	0	G	H

The Spectrum version looks a little different, because the AND command in Spectrum BASIC does not mask off bits we are uninterested in. It can only be used to give true/false results in expressions such as: IF X<3 AND Y=3 THEN. . . We therefore have to use a normal arithmetic subtraction operation to clear bit 2 in the Spectrum version of this routine.

By defining this general clear routine, we can also use it to clear the board before a game, by calling the routine with a zero parameter. This has been added in line 1400.

There is only one game procedure remaining before we can start writing the actual routines to process moves. This is the code that sets up the board at the beginning of the game with the appropriate handicap stones. As explained in the first instalment of this series (see page 1366), Go has a unique method of balancing skill levels by allowing the weaker player (who always takes Black) to place between two and nine stones on the board as the first move. These stones must be placed in specific positions, which are set up in the read handicaps routine (lines 600 to 750). If you ran the first part of the program given previously you'll have noticed that you were prompted to enter a handicap number at the end of the title_screen routine (but at the time the number was ignored). By adding line 1580 and the routine from lines 1630 to 1690, the handicap stones will now be

Module Two

BBC Micro

```
 270 PROCread_handicaps
 590 :
 600 DEF PROCread_handicaps
 610 LOCAL L%
 620 DIM hncp% 43
 630 RESTORE 670
 640 FOR L% = 0 TO 43
 650   READ hncp%?L%
 660   NEXT
 670 DATA &44,&CC
 680 DATA &44,&CC,&4C
 690 DATA &44,&CC,&4C,&C4
 700 DATA &44,&CC,&4C,&C4,&88
 710 DATA &44,&CC,&4C,&C4,&84,&8C
 720 DATA &44,&CC,&4C,&C4,&84,&8C,&88
 730 DATA &44,&CC,&4C,&C4,&84,&8C,&48,&
C8
 740 DATA &44,&CC,&4C,&C4,&84,&8C,&48,&
C8,&88
 750 ENDPROC
 760 :
 770 REM*******************************
1400 PROCclear(0)
1580 PROChandicap(hand%)
1620 :
1630 DEF PROChandicap(hand%)
1640 LOCAL L%,P%,Q%
1650 Q%=INT((hand%-2)/2*(hand%+1)+0.5)
1660 FOR L%=Q% TO Q%+hand%-1
1670   P%=hncp%?L% : board%?P%=black%
1680   NEXT
1690 ENDPROC
1700 :
1710 REM*******************************
4030 :
4040 DEF PROCcount(P%,C%)
4050 clib%=0 : cstn%=0
4070 PROCsearch(P%,C%)
4080 PROCclear(colour%)
4090 ENDPROC
4100 :
4110 REM*******************************
4120 :
4130 DEF PROCsearch(P%,C%)
4140 IF (P% AND 240)=0 OR (P% AND 15)=0
THEN ENDPROC
4150 IF (board%?P% AND colour%)=0 THEN
4250
4160 IF (board%?P% AND C%)=0 THEN ENDPR
OC
4170 IF (board%?P% AND marker%)>0 THEN
ENDPROC
4180 board%?P%=C%+marker%
4190 cstn%=cstn%+1
4200 PROCsearch(P%+dir%(1),C%)
4210 PROCsearch(P%+dir%(2),C%)
4220 PROCsearch(P%+dir%(3),C%)
4230 PROCsearch(P%+dir%(4),C%)
4240 ENDPROC
4250 IF (board%?P% AND liberty%)>0 THEN
ENDPROC
4260 board%?P%=liberty%
4270 clib%=clib%+1
4290 ENDPROC
4300 :
4310 REM*******************************
4320 :
4330 DEF PROCclear(M%)
4340 LOCAL L%
4350 FOR L%=0 TO 255
4360   board%?L%=board%?L% AND M%
4370   NEXT
4380 ENDPROC
4390 :
4400 REM*******************************
```

correctly placed; the DATA statements between lines 670 and 740 give the correct board positions, with each statement corresponding to a number of handicap stones between two and seven.

In the next instalment we'll add the routines required to allow you to play the game proper. Due to the structured way in which the program has been developed, we'll also be able to easily modify the program to allow two-player games, with the computer checking that nobody cheats.

Commodore 64

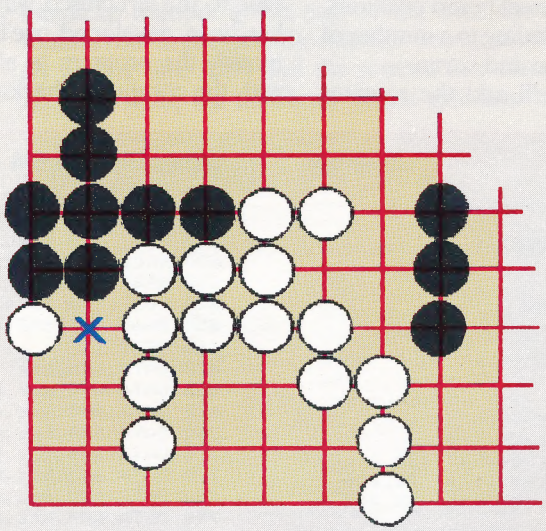
```
320 GOSUB 600
590 :
600 REM READ-HANDICAPS ROUTINE
620 HNCP=BOARD+512
640 FOR L=0 TO 43
650 READ H%:POKE HNCP+L,H%
660 NEXT
670 DATA  68,204
680 DATA  68,204, 76
690 DATA  68,204, 76,196
700 DATA  68,204, 76,196,136
710 DATA  68,204, 76,196,132,140
720 DATA  68,204, 76,196,132,140,136
730 DATA  68,204, 76,196,132,140, 72,200
740 DATA  68,204, 76,196,132,140, 72,200
,136
750 RETURN
760 :
770 REM*********************************
1400 MSK%=0:GOSUB 4330
1580 GOSUB 1630
1620 :
1630 REM HANDICAP ROUTINE
1650 Q%=INT((HND%-2)/2*(HND%+1)+0.5)
1660 FOR L=Q% TO Q%+HND%-1
1670 P%=PEEK(HNCP+L):POKE BOARD+P%,BLACK%
1680 NEXT
1690 RETURN
1700 :
1710 REM*********************************
4030 :
4040 REM COUNT ROUTINE
4050 CLIB%=0:CSTN%=0
4070 SP%=CP%:SC%=CC%:GOSUB 4130
4080 MSK%=COLOUR%:GOSUB 4330
4090 RETURN 0 THEN RETURN
4150 IF (PEEK(BOARD+SP%) AND COLOUR%)=0
GOTO 4250
4160 IF (PEEK(BOARD+SP%) AND SC%)=0 THEN
 RETURN
4170 IF (PEEK(BOARD+SP%) AND MARKER%)>0
THEN RETURN
4100 :
4110 REM*********************************
4120 :
4130 REM SEARCH ROUTINE
4140 IF (SP% AND 240)=0 OR (SP% AND 15)=
4180 POKE BOARD+SP%,SC%+MARKER%
4190 CSTN%=CSTN%+1
4195 SK%(STACK%)=SP%:STACK%=STACK%+1
4200 SP%=SK%(STACK%-1)+DIR%(1):GOSUB 413
0
4210 SP%=SK%(STACK%-1)+DIR%(2):GOSUB 413
0
4220 SP%=SK%(STACK%-1)+DIR%(3):GOSUB 413
0
4230 SP%=SK%(STACK%-1)+DIR%(4):GOSUB 413
0
4235 STACK%=STACK%-1:SP%=SK%(STACK%)
4240 RETURN
4250 IF (PEEK(BOARD+SP%) AND LIBERTY%)>0
 THEN RETURN
4260 POKE BOARD+SP%,LIBERTY%
4270 CLIB%=CLIB%+1
4290 RETURN
4300 :
4310 REM*********************************
4320 :
4330 REM CLEAR ROUTINE
4350 FOR L=0 TO 255
4360 POKE BOARD+L,PEEK(BOARD+L) AND MSK%
4370 NEXT
4380 RETURN
4390 :
4400 REM*********************************
```

Amstrad CPC 464/664

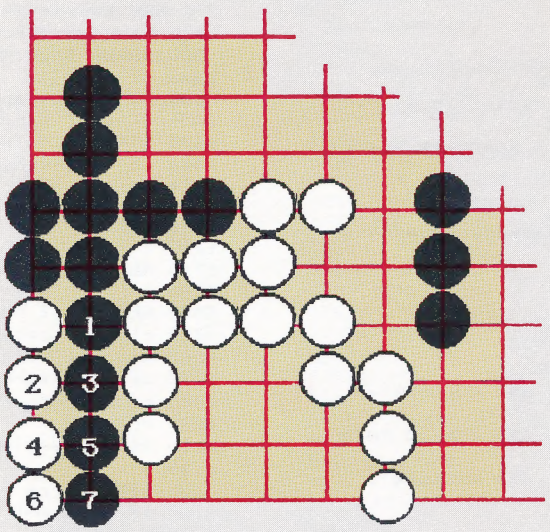
```
270 GOSUB 600:REM read handicaps
590 :
600 REM read handicaps routine
620 hncp=board+&200
630 RESTORE 670
640 FOR l%=o TO 43
650 READ h%:POKE(hncp+l%),h%
660 NEXT l%
670 DATA &44,&cc
680 DATA &44,&cc,&4c
690 DATA &44,&cc,&4c,&c4
700 DATA &44,&cc,&4c,&c4,&88
710 DATA &44,&cc,&4c,&c4,&84,&8c
720 DATA &44,&cc,&4c,&c4,&84,&8c,&88
730 DATA &44,&cc,&4c,&c4,&84,&8c,&48,&c8
740 DATA &44,&cc,&4c,&c4,&84,&8c,&48,&c8
,&88
750 RETURN
760 :
770 REM **********************
1400 mask%=0:GOSUB 4330:REM clear routine
1580 GOSUB 1630:REM hahdicap routine
1620 :
1630 REM handicap routine
1650 q%=INT((hand%-2)/2*(hand%+1)+0.5)
1660 FOR l%=q% TO q%+hand%-1
1670 LET p%=PEEK(hncp+l%):POKE(board+p%)
,black%
1680 NEXT l%
1690 RETURN
1700 :
1710 REM **********************
4030 :
4040 REM count routine
4050 clib%=0:cstn%=0
4060 cloc%(1)=0:cloc%(2)=0
4070 sp%=cp%:sc%=cc%:GOSUB 4130:REM search
4080 mask%=colour%:GOSUB 4330:REM clear
4090 RETURN
4110 REM **********************
4120 :
4130 REM search routine
4140 IF (sp% AND 240)=0 OR (sp% AND 15)=
0 THEN RETURN
4150 IF (PEEK(board+sp%)AND colour%)=0 T
HEN 4250
4160 IF (PEEK(board+sp%)AND sc%)=0 THEN
RETURN
4170 IF (PEEK(board+sp%)AND marker%)>0 T
HEN RETURN
4180 POKE(board+sp%),sc%+marker%
4190 cstn%=cstn%+1
4195 s(stack%)=sp%:stack%=stack%+1
4200 sp%=s(stack%-1)+dir%(1):GOSUB 4130
4210 sp%=s(stack%-1)+dir%(2):GOSUB 4130
4220 sp%=s(stack%-1)+dir%(3):GOSUB 4130
4230 sp%=s(stack%-1)+dir%(4):GOSUB 4130
4235 stack%=stack%-1:sp%=s(stack%)
4240 RETURN
4250 IF (PEEK(board+sp%) AND liberty%)>0
 THEN RETURN
4260 POKE(board+sp%),liberty%
4270 clib%=clib%+1
4290 RETURN
4300 :
4310 REM ********************
4320 :
4330 REM clear routine
4350 FOR l%=0 TO 255
4360 POKE(board+l%),(PEEK(board+l%) AND
mask%)
4370 NEXT l%
4380 RETURN
4390 :
4400 REM ********************
```

Sinclair Spectrum

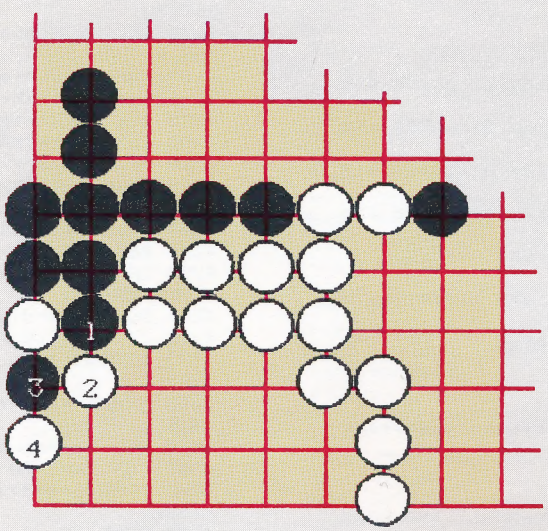
```
 270 GO SUB 600
 590:
 600 REM read-handicaps routine
 620 LET hncp=board+512
 630 RESTORE 670
 640 FOR l=0 TO 43
 650 READ h: POKE hncp+l,h
 660 NEXT l
 670 DATA  68,204
 680 DATA  68,204, 76
 690 DATA  68,204, 76,196
 700 DATA  68,204, 76,196,136
 710 DATA  68,204, 76,196,132,140
 720 DATA  68,204, 76,196,132,14
0,136
 730 DATA  68,204, 76,196,132,14
0, 72,200
 740 DATA  68,204, 76,196,132,14
0, 72,200,136
 750 RETURN
 760:
 770 REM ************************
1400 LET mask=0: GO SUB 4330
1580 GO SUB 1630
1620:
1630 REM handicap routine
1650 LET q=INT ((hand-2)/2*(hand
+1)+0.5)
1660 FOR l=q TO q+hand-1
1670 LET p=PEEK (hncp+l): POKE b
oard+p,black
1680 NEXT l
1690 RETURN
1700:
1710 REM ************************
4030:
4040 REM count routine
4050 LET clib=0: LET cstn=0
4070 LET sp=cp: LET sc=cc: GO SU
B 4130
4080 LET mask=colour: GO SUB 4330
4090 RETURN
4100:
4110 REM ************************
4120:
4130 REM search routine
4140 IF INT (sp/16)=0 OR sp-16*I
NT (sp/16)=0 THEN  RETURN
4150 IF PEEK (board+sp)=liberty
OR PEEK (board+sp)=0 THEN  GO TO
 4250
4160 IF PEEK (board+sp)=colour-s
c THEN  RETURN
4170 IF PEEK (board+sp)>colour T
HEN  RETURN
4180 POKE board+sp,sc+marker
4190 LET cstn=cstn+1
4195 LET s(stack)=sp: LET stack=
stack+1
4200 LET sp=s(stack-1)+d(1): GO
SUB 4130
4230 LET sp=s(stack-1)+d(4): GO
SUB 4130
4235 LET stack=stack-1: LET sp=s
(stack)
4240 RETURN
4250 IF PEEK (board+sp)>colour T
HEN  RETURN
4260 POKE board+sp,liberty
4270 LET clib=clib+1
4290 RETURN
4300:
4310 REM ************************
4320:
4330 REM clear routine
4350 FOR l=0 TO 255
4360 IF PEEK (board+l)>mask THEN
  POKE (board+l),PEEK (board+l)-
mask-1: GO TO 4360
4370 NEXT l
4380 RETURN
4390:
4400 REM ************************
```



Territorial Advantage
In this situation White has captured the bottom left corner of the board. However, this territory is not yet completely safe. The key point here is marked with a cross



Running For Cover
If Black is allowed to play on this key point then this move is called an 'Atari' on the white stone at the edge of the board. White may try to 'run' into the corner, but cannot avoid eventual capture by Black



Alternative Ploy
Rather than run into the corner, White could sacrifice the stone at the edge of the board, and in so doing stop Black's thrust into its territory by playing as shown

CAROLINE CLAYTON ON THE MACINTOSH

CRISPIN THOMAS

Read All About It
The Oberon Omni-reader is the first optical character recognition (OCR) device that is both reliable and within the price range of small to medium-sized businesses. Text from a printed page can be read by the Omni-reader and transferred directly into a computer's word processing software via an RS232C serial interface. This enables the text to be edited on screen without having to type in the entire document. At present, Oberon has provided software only for the most popular business micros although any computer with a suitable RS232C-compatible interface can use the system

AN EYE FOR DETAIL

Although they have been around since 1955, optical character recognition (OCR) devices have only recently become practical and reliable. The Omni-reader, from Oberon International, is one of the newer microprocessor-based OCR's, which can read printed text into a microcomputer fitted with an RS232C interface.

Over the past few years, there has been much speculation about the concept and development of the 'paperless office'. The idea stems from the fact that since data can be directly transferred from one computer to another, there is theoretically no need for the information to be on paper. However, the predictions concerning this ideal situation have proved to be somewhat premature. Most of us still find using pen and paper the most convenient method of noting down information. Also, data still has to be typed manually into the computer during the 'transitional' phase between today's office and the 'paperless office'. To speed up this process, optical character recognition (OCR) devices, which first appeared in 1955, read text from a printed page into a computer. However, they have only recently become reliable and practical propositions for business use.

The Omni-reader from Oberon International is one of the newer microprocessor-based OCRs appearing for business micros. Although the software is initially available only for such machines as the IBM PC and its compatibles, the Apricot range and the Apple Macintosh, it's conceivable to use the device with any micro fitted with a suitable RS232C interface.

The Omni-reader consists of a plastic document tablet with a horizontal rule that slides along a vertical bar on the left side of the tablet. The optical reader itself (a hand-held device with two buttons and an LED on top) is mounted on the rule and may therefore be moved to and fro across a document laid on the tablet, in very much the same manner as with an xy plotter.

A number of different functions and typefaces can be selected, which are flagged by LEDs along the top of the tablet. There are only four typefaces, or 'founts', which the Omni-reader can currently scan. These are Courier 10, Courier 12, Letter Gothic 12 and Prestige Elite 12, which are the four most popular founts used by daisywheel printers and electronic typewriters.

Inside the optical reader are two infra-red light sources on either side of a lens that focuses the image on the light detector circuitry. The detector can only read characters typed with carbon-based inks or photocopier toner (most of which is also carbon-based), creating some advantages and disadvantages. On the positive side, it means that ink from ballpoint pens and other non-carbon-based inks are 'transparent' to the detector. Thus appropriate text can still be read even though it

may have been marked with a pen or a rubber stamp. Furthermore, it also means that most coloured papers will not affect the operation of the Omni-reader. However, the sytem's design limitations are evident in that pencil marks will have to be completely erased if the text is to be read properly.

Attached to the optical reader is a cable that slots into the back of the Omni-reader tablet. Also along the back are the external power supply socket, a standard 25-pin RS232C D socket and two sets of DIP switches — one to set the baud rate and another that offers functions such as 'handshaking' and character spacing.

Ways Of Seeing
These four LEDs indicate additional functions. For example, POORCOPY allows badly printed text to be read, while NUMERIC reads numbers only
Which Type?
The four LEDs here indicate each of the four founts that are available to the device and are used in conjunction with the TYPEFACE command
1 C10
2 C12
3 LG
4 PE
X
Command Performance
These are commands that can be read by the Omni-reader. Note the two squares before the commands. This tells the Omni-reader that what follows is a command and should not be printed
TEST SHEET
THE OMNI-READER is a revolutionary new Text Reader or Optical Character Reader.
The quick brown fox can jump from the OMNI-READER into
over the lazy dog.
The Ruling Part
The rule allows the line of text to be positioned accurately within the scanning window. Note the coded strip at the bottom of the window to tell the OCR which way it is scanning
OMNI-READE
Outer Reaches
These two red arrows show the furthest limit of the paper. Beyond these arrows the OCR may not scan effectively

To read text into a computer, you first place the 'hard copy' onto the tablet, and then align a window, cut into the rule, over the line of text that is to be read. The button on the optical reader is pressed and the unit is moved along the rule, thereby scanning the text through the window. After about a second or so, if the Omni-reader has successfully read the line, the LED will flash once, a single beep will sound, and the line will appear on screen. If the reader has not read the line correctly, it will flash and beep twice. The user can then choose either to re-submit the line by making another pass across the screen (pressing the lower button and trying again), or tell the computer to accept the line by pressing the top button.

The Omni-reader compares the pattern that it reads with the list of characters it has for that particular typeface. The fount characters are held as bit-mapped templates within the Omni-reader's memory and the unit picks up characters by constantly scanning the image focussed through the lens. This image is then divided into 50 'slices',

OMNI-READER

PRICE

£914.25 inc VAT. This includes an interface cable, software, half a day's training and on-site maintenance for a year

INTERFACES

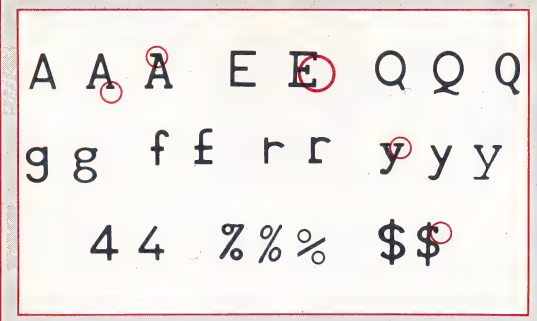
RS232C-standard interface with adjustable baud rate

STRENGTHS

The Omni-reader is fast and accurate and can input copy to a computer much faster than a typist

WEAKNESSES

Various hardware limitations restrict accurate reading to a limited number of typefaces and sizes. Because the ink has to be carbon-based, the scope of the device is further restricted

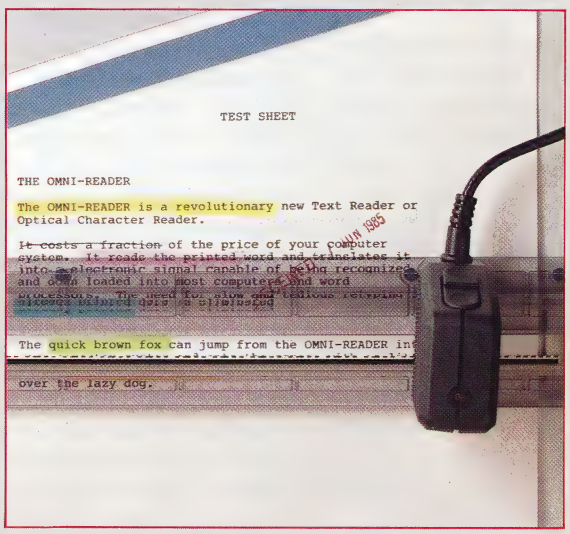


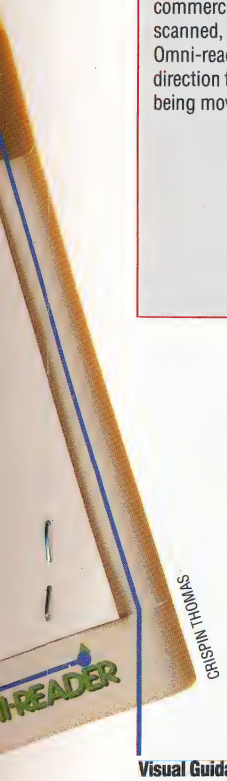
Dubious Characters

Although to the casual observer it might appear that all typefaces look much the same, there are certain key differences that make the electronic detection of characters difficult. Primary amongst these are the serifs, the extensions to the characters marked by circles. Because the Omni-reader has fixed templates in its memory, anything that fails to exactly match the character on the printed page will be discarded. However, even if we could provide some kind of 'fuzzy matching' to the character, some letters such as the examples of Q and g, are very different

Window To The Word

In order for the optical character recognition device to work efficiently, the text must be positioned exactly in the centre of the window. At the bottom of the scanning window is a series of blocks, rather like those seen in commercial bar codes. When scanned, these indicate to the Omni-reader in which direction the OCR head is being moved


TEST SHEET
THE OMNI-READER
The OMNI-READER is a revolutionary new Text Reader or Optical Character Reader.
The quick brown fox can jump from the OMNI-READER in
over the lazy dog.



Visual Guidance

The optical character reader is guided across the line of text by hand. At the end of a line, the LED on top will flash. One flash indicates a correct reading while two warns that the effort was not completely successful. A continuous light means that the device has failed to read the entire line

in both horizontal and vertical directions.

As it moves across the line of text, the Omni-reader is looking for entire characters underneath the head. It will only process a character that is both in the centre of its field of view and not cut by any of the boundaries of the area being scanned. (This is the reason why the character is scanned both vertically and horizontally.) The character is then sent to be matched with the characters in the device's memory.

When the Omni-reader finds a match, the software will then send the equivalent ASCII code to the computer via the RS232C interface and the character will apprear on the screen. Because the Omni-reader has to be able to find an exact match with the template held in memory, the text has to be aligned correctly within the scanning window. Therefore, if the descenders (letters like y, g, p and q, which have parts of the character below the 'line') are outside the window, they might be mistaken for something else — for example, a p might be mistaken for an o.

Alternatively, if the window is too far down, the optical scanner might pick up the tops of the characters on the line below and try to interpret them as well. Problems will also arise if the user tries to input text that is either too large or too small to fit comfortably within the scanning area. Oberon recommends that only 10 or 12-point characters (four of five characters per centimetre) be used. Anything outside this range is liable to generate an unacceptable level of error.

Within these restrictions, the Omni-reader in fact works surprisingly well. The optimum scan time for an A4 line of text is between 0.5 and 1.5 seconds, which is about how long it takes to make a smooth slide across the line. Although the novice user might have some initial difficulty lining the text correctly, it does not take very long to be able to judge the spacing; within half an hour you should be consistently producing error-free lines.

OPTIONS AND COMMANDS

In order to assist you in trying to read poorer quality typefaces, some options have been added. These options are fed into the Omni-reader by passing the optical reader over written commands — four of these are placed at the top of the tablet and the rest are listed in the manual. Each command requires two solid squares to precede it so that it is read as such, and not as a word to be displayed on screen.

Of the four commands that are set on the tablet, the most commonly used will be TYPEFACE, which enables you to choose the fount that will be read. NUMERIC removes the alphabetic characters from the fount and will only read numbers, thus speeding up the reading process. This option is perhaps one of the most useful applications of the Omni-reader since it is when copying figures that most undetected errors are usually made.

Direct input is a neat method of avoiding this problem. To deal with poorer quality printing, the option POORCOPY can be utilised, which slows down the read rate of the Omni-reader so that it can obtain a clearer image of the character under the optical reader. It is particularly useful when reading type generated from cloth ribbon systems where the carbon is less dense.

The Omni-reader software consists of a pair of driver programs that reside in the system area of memory. This means that applications programs can be loaded into the computer at the same time. Therefore, data read from a piece of paper can be input directly into a word processing program such as WordStar or, if you are reading in figures, can be read directly into the columns of a spreadsheet for instant calculations.

The price of the Omni-reader is £914, including software and a connecting cable to the computer of your choice. Also included is half a day's training and free on-site maintenance for a year. By any standard this is not particularly inexpensive. However, for a business that receives a lot of typewritten copy, such as a newspaper office, investing in an Omni-reader will definitely be cheaper in the long run than employing a full-time copy-typist.

PERFORMANCE PIECE

We conclude our MIDI interface series by considering further the techniques involved in writing MIDI software. Also, we include as Assembly language program that enables the BBC Micro to act as a digital recorder and playback device.

In the previous instalment, we examined the differences between normal analogue tape recording and the recording of digital MIDI data. We saw how these differences can be exploited to minimise the use of computer memory by using timing messages.

The digital recording and playback application provides us with a good example of some common problems we will often encounter when programming for MIDI. The program has to be able to receive and transmit MIDI data via the hardware interface designed earlier (see pages 1343 and 1368) and, moreover, this data must be received and transmitted in real-time. In other words, the program must run fast enough to accept, process and store one MIDI message before the next message arrives.

In addition, because we are recording in real-time, we must have some reliable timing source within the computer that will synchronise the program's actions in record and playback modes, and also record timing bytes within the memory array used to store the recorded sequence.

Most applications which require regular updating are interrupt-driven — that is to say, they use the IRQ interrupt signals to regularly execute particular sections of program code. However, for this application, the interval between successive IRQs may well be longer that the time between two successive MIDI messages. When the program is in record mode, this could prove disastrous, as whole MIDI messages could be missed, producing very odd effects when the piece is replayed. For example, if a 'note off' message was missed during the recording phase, that note, once triggered, would play on endlessly. We must therefore disable IRQ interrupts and seek another timing source for our program.

MIDI TIMING

The I/O chips used by the Commodore 64 and BBC Micro are similar and each has an on-board 16-bit timer register, which can be directly programmed to count down from a preset value to zero. On reaching zero, an interrupt flag is set and the timer is reloaded with its initial value. When operating in this way, the timer is said to be operating in 'free-run mode'.

The program uses this timer as follows: the subroutine check checks to see if the timer has 'timed-out' by interrogating the CIA/VIA chip's IRQ flag register and sets the Z flag in the processor status register accordingly. The recording and playback sections of the program both call the check subroutine and use BNE or BEQ instructions on return from the routine, branching according to the state of the Z flag.

The check routine also scans the keyboard to see if the Space bar has been pressed to stop the sequence. On the Commodore 64, this check is peformed, in the absence of the normal IRQ-driven keyboard scan, by checking the last row of keys on the keyboard. The BBC Micro version checks the keyboard via OSBYTE call &79. As long as we ensure that the check routine is called at least once between successive time-outs, this will provide us with a reliable method of timing the reception and transmission of MIDI messages

SYSTEM MESSAGES

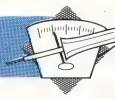
It is not necessary, or even desirable, to record every message sent on MIDI. Since the Recorder/Playback program generates its own timing, system real-time messages are irrelevant and must be ignored if received. (Most keyboard instruments will be incapable of transmitting these codes unless they have a built-in sequencer.)

System common and system exclusive messages may be inadvertently transmitted and must also be ignored. However, these messages may consist of any number of bytes and so a flag must be used to instruct the recorder to ignore all data until reception of a channel status byte occurs. This is indicated by a value of $FF in the X register. Other values held in the X register act as counters representing the number of data bytes to be received before the current channel message is complete.

Possible Improvements

This simple program is essentially an illustration of the type of problems encountered when processing real-time MIDI data. A number of possible improvements can be made to the program, requiring differing amounts of programming effort.

1. Stopping the playback in mid-sequence will, in general, leave a number of notes sounding indefinitely because they have not received 'note off' messages.
2. No provision is made for synchronisation of other devices, such as drum machines and tape recorders.
3. No editing of the sequence is possible.
4. Tempo of record and playback is fixed.
5. No overdubbing or 'punching-in/out' is allowed.
6. Recording is not possible while simultaneously monitoring another sequence.
7. If the MIDI cable is disconnected during the message, the program will hang up.
8. No provision for saving and loading recorded sequences to disk or tape is made



Record/Playback Program

```
  5 REM **   BBC MIDI PROJECT **
  6 REM ** DIGITAL RECORDING  **
  7 REM ** AND PLAYBACK       **
  8 *FX 14,6
 10 DIM start &2000
 20 FOR opt%=0 TO 3 STEP 3
 30 timer=&FE60
 40 osasci=&FFE3
 50 osrdch=&FFE0
 60  osbyte=&FFF4
 70 streg=&FEE0
 80 datreg=streg+1
 90 mem=&70
100 ptr=&72
110 P%=start
120 [
130  OPT opt%
140 .begin
150 JMP start1
160  .nobytes BRK
170  .clocks BRK
180  .freemem BRK
190  BRK
200 .start1
210 LDA #3
220 STA streg
230 LDA #&16
240 STA streg
250 .g10
260 LDA #&FF
270 STA lowmem
280 LDA #&40
290 STA timer+11
300 LDA #0
310 STA timer+4
320 LDA #&08
330 STA timer+5
340 .g20
350 LDA #0
360 JSR strout
370 .g21
380 JSR osrdch
390 CMP #ASC"E"
400 BNE g22
410 RTS
420 .g22
430  CMP #ASC"R"
440 BEQ g30
450 CMP #ASC"P"
460 BNE g21
470 .g30
480 PHA
490 LDA #lowmem MOD 256
500 STA mem
510 LDA #lowmem DIV 256
520 STA mem+1
530 LDA #frbytes MOD 256
540 STA freemem
550 LDA #frbytes DIV 256
560 STA freemem+1
570 SEI
580 LDY #0
590 PLA
600 CMP #ASC"R"
610 PHP
620  BEQ r00
630 .p00
640 LDA #2
650 JSR strout
660 .p05
670 JSR read
680 CMP #&FF
690 BNE p07
700 PLP
710 JMP c20
720 .p07
730 STA clocks
740 CMP #&F0
750 LDA clocks
760 BEQ p30
770 .p10
780 JSR check
790 BEQ p10
800 DEC clocks
810 BNE p10
820 BCS p05
830 .p30
840 JSR read
850 PHA
860 .p35
870 LDA streg
880 AND #2
890 BEQ p35
900 PLA
910 STA datreg
920 BPL p50
930 JSR getno
940 .p50
950 DEX
960 BNE p30
970 LDX nobytes
980 BPL p05
990 .r00
1000 LDA #1
1010 JSR strout
1020 STY clocks
1030 .r05
1040 LDX #&FF
1050 .r10
1060 JSR check
1070 BEQ r20
1080 INC clocks
1090 LDA clocks
1100 CMP #&F0
1110 BCC r20
1120 JSR store
1130 STY clocks
1140 .r20
1150 LDA streg
1160 AND #1
1170 BNE r40
1180 CPX #1
1190 BMI r10
1200 BPL r20
1210 .r40
1220 LDA datreg
1230 BMI r50
1240 CPX #0
1250 BMI r10
1260 BNE r80
1270 LDX nobytes
1280 BPL r60
1290 .r50
1300  CMP #&F8
1310  BCS r20
1320  CMP #&F0
1330 BCS r05
1340 JSR getno
1350 .r60
1360 PHA
1370 LDA clocks
1380 JSR store
1390 STY clocks
1400 PLA
1410 .r80
1420 JSR store
1430 DEX
1440 BEQ r10
1450 BNE r20
1460 .check
1470 STX &74
1480 PHP
1490 LDA #&79
1500 LDX #&E2
1510 JSR osbyte
1520 LDY #0
1530 PLP
1540 TXA
1550 BPL c40
1560 PLA
1570 PLA
1580 PLP
1590 BNE c20
1600 LDA #&FF
1610 STA (mem),Y
1620  .c20
1630 CLI
1640 JMP g20
1650 .c40
1660 LDX &74
1670 LDA timer+13
1680 AND #&40
1690 STA timer+13
1700 RTS
1710 .store
1720 STA (mem),Y
1730 INC freemem
1740 BNE point
1750 INC freemem+1
1760 BNE point
1770 PLA
1780 PLA
1790 PLA
1800 LDA #3
1810 JSR strout
1820 JMP c20
1830 .read
1840 LDA (mem),Y
1850 .point
1860 INC mem
1870 BNE p20
1880 INC mem+1
1890  .p20 RTS
1900 .getno
1910 LDX #2
1920 CMP #&C0
1930 BCC n10
1940 CMP #&E0
1950 BCS n10
1960 DEX
1970 .n10
1980 STX nobytes
1990 INX
2000 RTS
2010 .strout
2020 ASL A
2030 TAX
2040 LDA messtab,X
2050 STA ptr
2060 LDA messtab+1,X
2070 STA ptr+1
2080 LDY #0
2090 .m10
2100 LDA (ptr),Y
2110 JSR osasci
2120 INY
2130 CMP #&0D
2140 BNE m10
2150 LDY #0
2160 .m70 RTS
2170 ]
2180 mess0=P%
2190 $P%="R = record,P = play,E = exit"
2200 P%=P%+LEN($P%)+1
2210 mess1=P%
2220 $P%="recording"
2230 P%=P%+LEN($P%)+1
2240 mess2=P%
2250 $P%="playback"
2260 P%=P%+LEN($P%)+1
2270 mess3=P%
2280 $P%="out of memory"
2290 P%=P%+LEN($P%)+1
2300 lowmem=P%+8
2310 frbytes=&E000+(lowmem-start)
2320 [
2330 .messtab
2340 ]
2350  NEXT
2360 ?P%=mess0 MOD 256
2370 P%?1=mess0 DIV 256
2380 P%?2=mess1 MOD 256
2390 P%?3=mess1 DIV 256
2400 P%?4=mess2 MOD 256
2410 P%?5=mess2 DIV 256
2420 P%?6=mess3 MOD 256
2430 P%?7=mess3 DIV 256
2440 CALL begin
```

TINY TURTLE

As one of the better introductory computer languages, LOGO has gained a wide appeal in the educational market as well as with home micro users. Here, we look at Dr LOGO for the Amstrad computers and compare it with the original produced for the IBM PC.

When LOGO was first transferred from mainframe computers to eight-bit microcomputers, a number of restrictions and simplifications had to be made so that the language could run within the 64 Kbytes or less of memory that was available. As 16-bit micros began to appear with much larger memories, a number of enhanced versions of LOGO quickly became available.

Gary Kildall — the founder of Digital Research (the producers of the CP/M operating system) — was excited by LOGO's potential as a language and had produced his own version for the IBM PC, calling it Dr LOGO. This version has since been made available on a number of other 16-bit machines. The Apricot F1e (see page 1389), for example, comes supplied with Dr LOGO.

Digital Research has since adapted Dr LOGO for a number of eight-bit CP/M machines. Amstrad is providing such a version of Dr LOGO along with its disk pack for the CPC 464.

Dr LOGO, as implemented on the IBM PC, requires at least 192 Kbytes of memory to run. The disk is copy-protected, but a backup copy is provided. To run it, you just insert the disk and peform a system reset. The colour graphics resolution is 320 by 200 pixels with a choice of 16 background colours, together with a choice of four sets of three pen (foreground) colours.

With a colour graphics card on the PC, along with both a colour and a monochrome screen, Dr LOGO can produce graphics on the colour screen and text on the monochrome screen! However, if you only have one monitor, the two can be mixed on the same screen.

Dr LOGO is described in the manual as a superset of Apple LOGO and it in fact contains all the LOGO commands to be found there. These include the standard graphics, list processing and workspace management commands as well as error handling primitives, 'properties' (a way of assigning more than one value to an object) and 'packages' (a way of bundling procedures together as a group). Dr LOGO is 'case sensitive' and therefore LOGO commands must be entered in lower case.

The editor makes use of the IBM function keys but also recognises the same control codes used in Apple LOGO. If an error is found while running a program, typing ed will implement the editor with

LOGO Benchmarks

PROCEDURE		RESULTS			
		IBM PC (256K)	AMSTRAD CPC 464/664	COMMODORE 64	APRICOT F1e
RECUR	Level Reached	512	187	90	832
DEFVARS		1307	263	553	1606
ENDRECUR1		512	169	INFINITE	INFINITE
ENDRECUR2		151	129	65	72
LINES (with turtle shown)	Timing (To The Nearest Second)	9	53	13	10
LINES (with turtle hidden)		5	39	11	6
ASSIGN		25	22	6	20
LISTS		64	74	18	69
ARITH		80	138	36	77
RECURA		92	91	24	72

The Benchmark Procedures

The nine benchmark procedures are designed to test different features of LOGO and to show the strengths and weaknesses of each implementation of the language.

- RECUR measures the size of the stack
- DEFVARS measures the space available for defining variables and procedures
- ENDRECUR1/ENDRECUR2 determine whether end recursion has been efficiently implemented. An ideal implementation should allow infinite recursion. While ENDRECUR2 outputs its result and can therefore be considered as an operation, ENDRECUR1 is a command
- LINES measures the drawing speed. Two figures are given to illustrate the improvement in speed gained by hiding the turtle
- ASSIGN measures the speed with which variables can be assigned values
- LISTS measures the speed of the list processing operators
- measures the speed of the arithmetic functions
- measures the speed of recursive calls

the offending procedure in place, ready to edit. Unfortunately, there is no way of telling by looking at the screen when you are in the edit mode. Most other versions of LOGO indicate this by having a special line at the bottom of the screen.

NEW PRIMITIVES

Besides the usual list processing primitives, Dr LOGO has a number of new ones:

sort sorts a list into alphabetical order
shuffle puts the elements of a list in random order
piece enables you to select a part of a list

Dr LOGO adds a number of debugging tools:

watch enables you to see the effect of the procedure line by line
trace prints out the names of variables as they are called and as they are defined
debug splits the screen into two windows. The 'debug' window can be used to display information provided by trace and watch, while the 'program' window shows the output from the program

Comments can be added to procedures but they can then be removed from all procedures in the workspace, with noformat, if this is needed to provide extra space for running.

There are also a number of new procedure management primitives:

follows enables you to define the order in which procedures will be presented on the screen
potl displays names of procedures not called by any other procedures
poref <name> displays names of procedures that call the procedure <name>
pocall <name> displays names of procedures called by the procedure <name>

What these extra features amount to is an improved program development environment in which to create LOGO programs.

There are a number of weaknesses in Dr LOGO: there is no search and replace function in the editor, no method of interfacing to machine code and no file handling. The 300-page manual, however, is very clear and thorough. It has a tutorial introduction to LOGO as well as a reference manual that includes a separate page devoted to each primitive.

Clearly, Dr LOGO has a number of attractive extra features not found in smaller versions of the language, all of which results in an improved program development environment.

It is interesting to see how Dr LOGO compares, in terms of space and speed, with the eight-bit versions of the language. LOGO workspace is measured in 'nodes'. Asking Dr LOGO how many nodes it has free on starting the system gives a comforting answer of around 10,000 (eight-bit systems typically have from 2,000 to 3,000). So there is plenty of workspace for defining procedures and for running recursive procedures. However, end recursion — the case of a recursive procedure in which the recursive call is in the last line — is not implemented efficiently, and so programs that would run infinitely on Commodore or BBC machines, run out of space after a few hundred recursive calls with Dr LOGO.

As far as speed is concerned, the graphics are fast, but arithmetic and list processing are both performed more slowly than with other eight-bit versions of the language.

THE AMSTRAD VERSION

Dr LOGO is supplied with the Amstrad disk drive (see page 1209), and is on the opposite side of the CP/M disk. Dr LOGO on the Amstrad is modified to take advantage of the hardware it is running on. The graphics use the same four-pen system as on the Dr LOGO for the IBM PC, but the pen colours are now set by giving a list of numbers determining the proportions of red, green and blue required. The IBM version does not have an extensive range of sound commands, whereas the Amstrad version has env (for volume envelope), ent (for tone envelope) and release, which releases channels set to a hold state in a sound command.

The Amstrad version has most of the features of Apple LOGO — including properties and error handling, but not packages.

The debugging primitives and the extra procedure management features of Dr LOGO on the IBM PC have been left out, presumably because of the smaller memory capacity of the Amstrad. More serious, however, is that there is no define function and seemingly no way of deleting a file without exiting to CP/M and then coming back in again.

The editor is very slow to respond and you might find yourself typing ahead and losing the odd character. On the other hand, if an error is found in running a procedure, the editor, once entered, positions the cursor over the offending word in the procedure, which is a very useful feature indeed.

The Amstrad version provides limited access to machine code via .examine and .deposit (PEEK and POKE), which the IBM PC version does not have.

The documentation supplied is minimal, and you are referred to a tutorial and reference manual that has yet to be published. In the meantime, you will have to rely on the 25-page summary of the commands. Also, there appear to be some features of Dr LOGO that have been implemented but have not been mentioned.

A lot of the more exciting features of Dr LOGO for the IBM PC have disappeared in the Amstrad version, but we are still left with a good standard implementation of LOGO, with extras such as packaging and error handling.

On starting up, Dr LOGO claims to have 2,105 nodes, which is a typical figure for eight-bit systems. However, end recursion has not been efficiently implemented; and for graphics, arithmetic and list processing, this is one of the slowest eight-bit LOGOs, with most operations taking about three times as long as on the Commodore 64.

AND SO FORTH

FORTH is a programming language that was originally developed to control a radio telescope, and it is generally regarded as best suited to such control applications. Our introduction to a series on the language looks at FORTH's two key features — interactiveness and extendability.

Invented in the late 1960s by astronomer Charles Moore, FORTH was specifically designed to control a scientific device that needed precise positioning. The language's usefulness extends far beyond control applications, however, but as an introduction to its basic operating principles, it is instructive to look at how it can be used to control a telescope.

It would be nice to able to sit down at a computer terminal in the observatory, and — for example — type in:

```
30 DEGREES ELEVATION
```

at which point the telescope would move immediately and accurately. It would also be useful to be able to package and store useful sequences of such commands to save repetitive typing. For instance if the parked position has 90° elevation and 0° azimuth (these are terms describing the angle and position of the telescope), you could define a new command, PARK, as:

An Ace Idea
The Jupiter Ace was a bold attempt to market a machine offering FORTH, rather than BASIC, as the standard language. Sadly, the Ace's lack of colour facilities coupled with a small memory, the unfamiliarity of FORTH and strong competition from the Sinclair Spectrum left it at a severe disadvantage compared to other home micros. A second-hand machine, however, together with the original documentation (which was of a very high standard) would provide an excellent introduction to the language. The Jupiter Ace is now being marketed by Boldfield Computers Ltd, Cambridge.

IAN McKINNELL

```
0 DEGREES AZIMUTH 90 DEGREES ELEVATION
```

In summary, then, you would need:

- Elementary telescope commands.
- Ways of combining the elementary commands into more complex ones — telescope 'programs'.

This system could amount to a complete telescope programming language. It would, however, require an extra, rather special, property: it would have to accept commands — both the built-in commands like ELEVATION and any new ones — directly from the keyboard. Additionally, it would have to do this in as simple a manner as possible.

Imagine trying to fulfil these requirements in BASIC, which is quite good at letting you use its built-in commands, like PRINT and RUN, from the keyboard. Your new commands, however, will be a problem. Assuming you have already defined AZIMUTH and ELEVATION as GOSUB routines at lines 1000 and 1100, you might write a command for PARK as:

```
1200 REM PARK
1205 LET DEGREES = 0
1210 GOSUB 1000
1215 LET DEGREES = 90
1220 GOSUB 1100
1225 RETURN
```

Or you might be able to predefine a variable PARK by:

```
LET PARK = 1200
```

and subsequently issue the command (in some versions of BASIC):

```
GOSUB PARK
```

The best you are likely to be able to do is define the GOSUB routine as a procedure (such as in BBC BASIC) and type something like:

```
PROCPARK
```

Thus, whenever you call one of your own telescope routines, you have to enter additional commands, like GOSUB or PROC, as well as the name or line number of your routine.

An alternative is to write an interpreter, a program that takes telescope commands and executes them — something like:

```
100 INPUT C$
150 IF C$ = "PARK" GOSUB 1200: GOTO 100
```

But then you can no longer type in the BASIC commands (like PRINT) directly.

FORTH'S ESSENTIAL PROPERTIES

We can see now that the two essential properties for our telescope language will be:

- First, that it must be *interactive*. It must be able to execute your commands as soon as you type them in. This is like BASIC, LOGO, APL and PROLOG but unlike PASCAL, ALGOL, FORTRAN and COBOL. It is

also like the command end of operating systems, such as CP/M and Unix.

- Second, that it must be *extendable*. The format for calling commands you've defined yourself must be identical to that used for the built-in commands. Then, once you've defined your routines it will be as though you are using an extended, more powerful language, rather than the original language plus some new routines. This is like LOGO and the operating systems, but unlike all the other languages.

The concept of extendability is in fact even more powerful than it has appeared so far. If, for example, someone sells us a telescope control language and we extend it in minor ways to include useful routines for our own telescope, we will have extended an original 'general purpose telescope control language' to a new 'personalised telescope control language'. But if the extendability is to be fully realised, we should be able to begin with a 'general purpose extendable computer language' and extend that to a 'general purpose extendable computer language' and extend that to a 'general purpose telescope control language' (indeed, this could be extended to languages to control robots, scientific experiments, production lines or anything else that can be connected to a computer). Furthermore, what is controlled does not have to be physical. It could be an abstract concept internal to the computer, like the turtle on a turtle graphics screen, the records in a filing system or, in fact, anything you want handled by a program.

FORTH is the closest approach we have to a 'general purpose extendable computer language'. We have looked at the special problems — interactiveness and extendability — that it tries to solve, and in our next instalment we will begin to look at its solutions.

Forth Dialects

There are three main dialects of FORTH: fig (FORTH Interest Group) FORTH, FORTH-79 and FORTH-83. Most implementations are described as conforming with one of these three.

FORTH-79 and FORTH-83 are successive standards, and specify minimal sets of commands and routines that must be available. Individual implementations will almost certainly give you more, but if you write a program in strict FORTH-83, for example, it should run on all FORTH-83 implementations, even though it won't make full use of all the available facilities. FORTH-83 is the current standard and we shall conform with that. FORTH-79 is broadly the same, with just a few incompatibilities. It also has a lot of extra facilities, incorporated in the process of building up the implementation itself, which its users tend to treat as standard. Many FORTH-79 and FORTH-83 implementations are versions of figFORTH modified to make it conform with the standard

The Telescope And The Turtle

A comparison between FORTH and LOGO is instructive, as they are surprisingly similar. After all, both languages were originally designed to control physical objects — a telescope on the one hand, and a turtle on the other. Two fundamental operating concepts are common to both languages: a routine is called simply by entering its name, and new routines are defined using existing routines.

The main difference between the two languages results from the different expectations of their intended users. FORTH users demand speed of execution (so the language is not only interactive and extendable, but also efficient) and are prepared to put up with some awkwardness to help achieve this. LOGO, on the other hand, is primarily intended to be used by school children and allows for no such awkwardness. Consequently, LOGO runs much more slowly.

Let's compare the two more closely. FORTH doesn't have turtle graphics built into it as standard, but it does have facilities for handling the terminal screen. You can use these low-level facilities to define the turtle words FORWARD, RIGHT and so on — thus creating an 'extended' version of FORTH that includes turtle graphics. Consider these alternative ways of controlling the turtle in the two languages:

LOGO	**FORTH**
FORWARD 100	100 FORWARD

From this example, FORTH would seem to be doing things backwards. But when a FORTH routine needs parameters, you must calculate those parameters first and write them before the name of the routine. This could be regarded as a 'recipe book' technique — first gather the ingredients (in specified quantities) and then cook them.

Other correspondences with LOGO include the definition of new routines:

TO. . .END	: . . . ;

and loop structures:

REPEAT 4 [. . .]	40 DO . . . LOOP

Here's how we might combine the two in the definition of a SQUARE procedure:

```
TO SQUARE SIDE          : SQUARE
REPEAT 4[                 4 0 DO
  FORWARD SIDE              DUP FORWARD
  RIGHT 90                  90 RIGHT
  ]                       LOOP
                          DROP
END                     ;

SQUARE 100                100 SQUARE
```

DUP and DROP are 'stack manipulations' which enable the routines to access the parameter of SQUARE (DUP duplicates it, making a copy for FORWARD; DROP 'drops' it — in other words, throws it away so that it won't get in the way of the next calculation)

MARCUS WILSON-SMITH

Go Forth. . .

There are FORTH implementations for many home computers. The more readily available packages for the most popular micros include: Acornsoft FORTH (also available in ROM) and SkyWays MultiFORTH (with multitasking) for the BBC Micro; and Abersoft FORTH and Artic FORTH for the Sinclair Spectrum.

FORTH is the only computer language that was propagated by its users (who originally sent forth implementations of the language via User Clubs). There is a thriving FORTH Users' Club in London. The group meets at 7pm on the first Thursday of each month at the South Bank Polytechnic, Borough Road, London SE1. You can join (an annual subscription is £7) by writing to:

FORTH Interest Group
7 Wyndham Cresent
Woodleigh
Reading
RG5 3AY7719

SERIAL DRAMA

The Interface 1 provides a lot of useful routines for the Spectrum machine code programmer. We have already taken a general look at the facilities offered by the interface (see page 1398); now, we turn our attention to those hook codes that it uses to control the RS232 serial port.

Let's begin our discussion with a look at the serial interface, which is a nine-pin 270° RS232 socket (our diagram shows the function of each pin). The number of these lines that you will need to connect depends upon the complexity of your application, and a minimal RS232 link can be established by using pins 2, 3 and 7. If you do this, pins 4 and 5 should be connected to pin 9 to hold them at logic one. This prevents the Spectrum from 'hanging up' if whatever is on the other end of the link isn't yet ready to receive data from the computer. This has the disadvantage, however, of losing data if the other end of the link isn't able to receive it. It is better, on the whole, to use the full wiring.

A possible application of the RS232 link is to connect the Spectrum to a 'real' printer. Once the cable is wired up, the controlling software is simple to use. To send text to the printer, we set an appropriate baud rate from BASIC using FORMAT, and then execute the following code:

```
CLOSE #3
OPEN #3;"T"
```

This attaches stream 3, which is usually the ZX printer stream, to channel T, which is the text channel of the RS232 link. A full discussion of the use of 'text' and 'binary' streams is given in the Interface 1 manual. LLIST and LPRINT will now send data to the device attached to Interface 1.

Now let's consider the details of the operation of the serial interface. Data is transmitted with one start bit, eight data bits and two stops bits. We do not have to worry about this format, as the Spectrum operating system routines look after it for us. What is more important to us is the rate at which data is sent — the *baud rate*. The baud rate is usually set up by the FORMAT command, which restricts us to a certain range of baud rates. Although there is little loss of data integrity with an RS232 interface — even over a considerable distance — it is a good idea to use slower transmission rates with increasing distances between sender and receiver.

It is good practice to insert the Interface system variables using hook code 49 before using any of the following techniques. There are two hook codes concerned with RS232, and three system variables. Let's look at the variables first.

- BAUD (at locations 23747 and 23748) is a two-byte variable, which holds a value that determines the baud rate to be used for the transmission and reception of data. This presents us with one of the limiting features of the Spectrum serial interface — its inability to simultaneously have different baud rates for send and receive.
- SERFL (at location 23751) is a flag byte, which is used by the OS on receive only. If it is set to one, then it indicates that there is still a byte available to be read.

SERBYT (at location 23752) acts as a one-byte input buffer for receive. It is occasionally possible for a byte to be received and stored in this location after you've discontinued communication. Problems can arise with the use of the latter two variables — we'll look at these later.

A fourth system variable — IOBORD (at location 23750) — is not strictly concerned with the serial interface, but holds the colour of the screen border to be used in I/O operations. You can therefore set it to hold any colour you wish for the screen border during these operations.

When we want to use the RS232 interface, then the first thing to do is to set up a baud rate. This is done by putting an appropriate value in BAUD — indeed, this is all that FORMAT does. The value to be used for a particular baud rate is given by:

```
value = (3500000/(26*baud rate))-2
```

Baud Rates Using Interface 1

The RS232 standard is extensively used in communications, particularly when data transmission over long distances is required, and the 'baud rate' provides an indication of the number of data-bits per second transmitted. Applying the formula indicated in the text to some of the more common baud rates results in the following values:

Baud Rate	Value
50	2690
110	1221
300	446
600	222
1200	110
2400	54
4800	28
9600	12
19200	5

The value for the desired baud rate is then POKEd into the system variable BAUD at addresses 23747 and 23748. The single system variable means, unfortunately, that the same baud rate must be used for both sending and receiving data. (For further information on baud rates, see page 903.)

The following piece of code shows how these values are used. To set a baud rate of 300 baud:

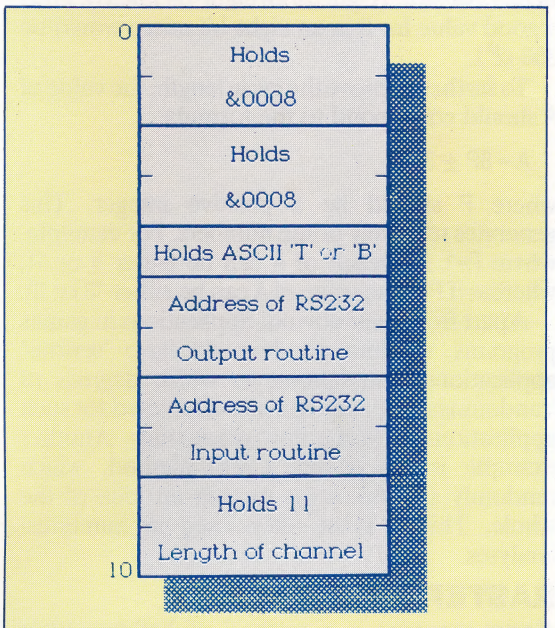
```
LD    HL,446
LD    (23747), HL
RET
```

By using this equation, it's possible to set up the serial interface with a non-standard baud rate from machine code, but this is probably of minimal use unless you are communicating with another Spectrum via the RS232. The Baud Rate table gives a variety of baud rate values and the necessary routine for implementing them.

Now we can consider the way in which data is transferred across the link. In BASIC, the act of OPENing a stream for use with the RS232 interface generates a channel of the form shown in the RS232 Channel Structure diagram. This channel is placed in the Channel Area of memory. Byte 4 of this channel has 128 added to its value if the channel has been created implicitly (that is, using SAVE*, MOVE, FORMAT, and so on — see page 1418).

However, this is only of academic interest to us when we come to use the RS232 routines from machine code. You will note that there is no buffer associated with such a channel. We write and read data by calling up the relevant hook code, which in turn calls the routine that has its address in either channel locations 5 and 6 (for transmission) or 7 and 8 (for reception). When we use a hook code, no channel is set up. The two RS232 hook codes are:

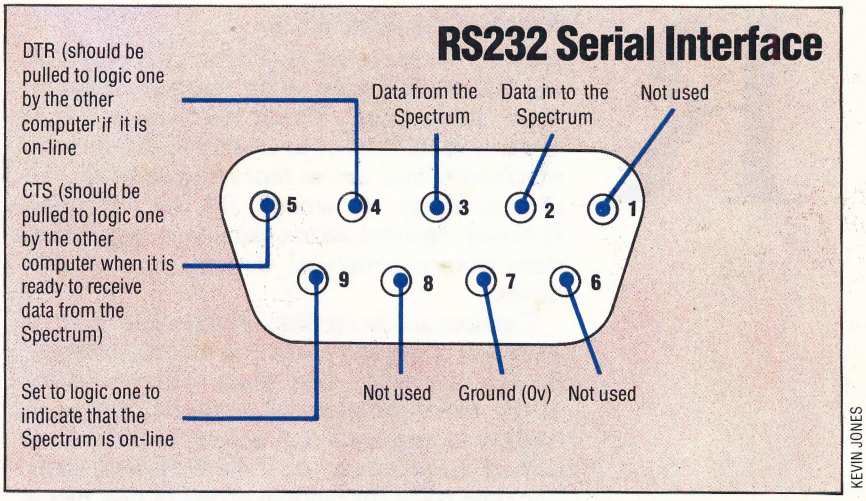
- *Hook Code 29:* When called, this returns a single byte from the serial interface in the A register, if one is available (in which case, the carry flag will be set to one). If there isn't a byte available, then A will contain zero and the carry flag will be clear.



CAROLINE CLAYTON

RS232 Channel Structure
Note that there is no buffer accociated with the channel. All data to be sent or received is passed in the A register via hook codes 29 and 30

- *Hook Code 30:* This transfers the byte in the A register to the serial interface for transmission at the current baud rate. When we use this code, all bytes are transmitted without modification; there is no difference between the B and T streams at the machine code level.


RS232 Serial Interface
DTR (should be pulled to logic one by the other computer if it is on-line
CTS (should be pulled to logic one by the other computer when it is ready to receive data from the Spectrum)
Set to logic one to indicate that the Spectrum is on-line
Data from the Spectrum
Data in to the Spectrum
Not used
5
4
3
2
1
9
8
7
6
Not used
Ground (0v)
Not used

KEVIN JONES

It is at this point that the problems regarding SERBYT and SERFL can arise. If the hook code routine finds that SERFL is set to one, then it will get a byte from SERBYT, even if that byte has been 'hiding' in SERBYT since the last bout of serial communication was completed. This 'rogue byte' needs to be dealt with if it appears at the beginning of a series of received bytes; all those following will still be received, but their meanings may now be altered. The easiest way to get rid of what is in SERBYT is simply to set SERFL to zero before attempting to read anything from the serial interface. This can be done with a simple piece of code:

```
XOR     A
LD      (23751),A
```

That is the only real complication that most people will find when using the serial interface — although, should you be unable to get anything at all out of it, it is always worth swapping the connections to pins 2 and 3 on the Interface 1 plug, in case they were wrongly connected in the first place. The baud rate generated, though not always precisely defined, is correct to within the tolerance required by the RS232 specification.

As an example of using the RS232 interface, the following piece of code transmits 255 bytes of data down the serial interface:

```
        ;send 255 bytes of data down serial interface
        ;
CF              rst  #8                 ;set up...
31              defb 49                 ;...I/face 1 variables
216E00          ld   hl,110             ;value for 1200 Baud
22C35C          ld   (23747),hl         ;set baud rate
06FF            ld   b,255              ;set counter
C5      loop:   push bc                 ;save counter
3E00            ld   a,DATA             ;user supplies routine..
CF              rst  #8                 ;...to get data in a reg
1E              defb 30                 ;send the byte in a
C1              pop  bc                 ;restore counter
10F8            djnz loop
C9              ret
```

In many ways, the RS232 link is the easiest of the Interface 1 functions to use. There is no channel information to worry about from machine code, and we simply have two calls — transmit and receive — to deal with.

Means Of Communication
The Interface 1 serial port provides a means of communicating with other serial devices and uses a standard nine-pin D socket. The only serious drawback of the Spectrum serial interface is its inability to use different baud rates for receive and transmit

R

RANDOM ACCESS

Random access refers to a type of memory in which all the addresses within the system have an equal access time. Within the semiconductor memory itself, it refers to ROM and RAM chips, which both have access times (not including the address time) of around 0.1 microseconds. However, random access can also be applied to mass storage peripheral devices, such as disk drives.

Random access on disks requires that there be some sort of directory listing on the disk to inform the disk operating system where each component of the file to be accessed is stored. This is in contrast to 'sequential disk access', which looks at each of the sectors to see if the file is held there. The time taken for random access of disk files is generally between eight and 30 milliseconds.

RANDOM NUMBER

As its name suggests, a *random number* is a number that is generated at random from within an allowed range. Generating real random numbers is a notoriously difficult process as computers, by their very nature, cannot behave randomly. However, certain machines have been devised specifically to produce truly random numbers. These machines generally use the movement of subatomic particles in order to generate their numbers — by counting the intervals between alpha waves (which are emitted at random) appearing from radioactive sources, for example, or by the random diffraction of electrons in a gas sensed by transducers.

Various methods have been devised to generate such numbers arithmetically. Such methods are said to generate 'pseudo-random numbers', which, although not truly random (as the sequence is predictable and cyclic), are nevertheless indistinguishable from real random numbers in most applications.

To generate random numbers, we first need a 'seed' value, which is often taken from the system clock. From there, we can choose from a variety of procedures that will produce a series of random numbers. One of the more common is the 'middle square' method put forward by John von Neumann. In this system, numbers are squared and the middle four digits are used as both a random number and the seed for the next.

The problem with this method, as with all pseudo-random number generating systems, is that after a while the numbers begin to repeat themselves or fall to zero. A slightly better method, in which the cycle length is somewhat longer, is the 'middle product' method. Here, a pair of seed values will be required, and both are multiplied together to form a number. The middle four digits of the result are taken as the random number. To generate the next number, the first number is abandoned and the two new seeds are multiplied together.

To produce really long cycles, however, another technique is used, called the 'congruence method'. This involves placing carefully chosen constants within a formula, which in the case of the 'multiplicative congruence' is:

$$Xn+1=A/-Xn(MOD\ B)$$

where Xn is the current random number, and A and B are constants. In order to ensure that the cycle length is as long as possible, X0 (the seed value), A and B should be chosen with care. X0 should, for example, have no factors in common with the modulus B. It is also advisable, when using binary arithmetic, that $B=2^W$, where W is the number of bits available (eight on an eight-bit micro). Thus a good value for B in an eight-bit micro would be 256 (2^8).

To further improve the cycle length, the value of A should correspond to the formula:

$$A = 8P \pm 3$$

where P should be a positive integer. This generates relatively prime numbers. For example, where P=1 the result of A can be either 5 or 11, whereas if P=2 the value of A can be either 13 or 19.

Apart from their obvious applications in games programs, random numbers do have 'serious' applications. Simulations, in which degrees of 'randomness' are required, are often used for such applications as traffic flow predictions. Another example is the Monte Carlo method, which randomly samples a set to gain an idea of the whole. This is most often used in numerical analyses.

RASTER SCANNING

Raster scanning is a method of building up a picture on a VDU screen and is most commonly used in standard television screens. The electron gun at the base of the cathode-ray tube draws a series of lines from the top to the bottom of the screen, turning on each of the screen's electroflourescent points in turn. When the final line on the screen has been drawn, the gun returns to the top and 'refreshes' the picture.

Raster scanning is used by the computer for a variety of applications, not only for producing graphics displays. It is often used by light pens (see page 969) in order to fix their positions.

COURTESY OF NATIONAL SAVINGS

A Nice Little ERNIE
One of the most famous computerised random number generators is ERNIE (Electronic Random Number Indicator Equipment), which is used by the Department for National Savings to select winning premium bond numbers. In order to generate true random numbers, the machine uses electron diffraction. In this method, a beam of electrons is passed through a gas, and the resulting random diffraction of electrons is used to generate a number

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 61 TO 72

A

ACIA chip 1343-1345
ACIA registers 1369
Acquisition module 1303
Adams, Douglas 1400
ADSR 1238
Aleksander, Igor 1382
Alpha-beta algorithm 1269, 1281
Alphacom 32 1261-1263
Alvey program 1293
Amstrad CPC 464/664 1207-1208, 1209-1211, 1264-1265, 1405
Go game 1407-1408, 1427
Amstrad DDI-1 disk drive **1209-1211**
Apricot F1e **1389-1391**
Apricot portable 1280
Archon 1260
Artificial intelligence **1221-1223, 1241-1243, 1268-1271, 1281-1283, 1301-1303, 1321-1323, 1348-1351, 1361-1364, 1381-1383, 1401-1403, 1421-1424**
ASM 1285
Atari 130XE **1309-1311**

B

Backgammon 1282-1283
Backed-up value 1282
Backup files 1304
Banana interface **1290-1291**
Basic Disk Operating System (BDOS) 1324-1325, 1346-1347
Basic Input/Output System (BIOS) 1324-1325, 1346-1347
BBC Micro
Cubic Splines program 1227
digital tracer 1253-1255, 1274-1275, 1286-1288, 1306-1307
Go game 1393-1395, 1426
MIDI interface 1335-1337, 1343-1345, 1368-1371, 1432-1433
Numbers game 1271
simulation game 1212-1214, 1232-1233, 1244-1245, 1266-1267, 1294-1295, 1314-1316, 1338-1339, 1354-1356
BEAGLE system 1321-1323
Behavioral Engineering 1202-1203
Berliner, Hans 1281
Blocks World 1422
Bosom, Rosa 1421
Branching factor 1282
Brother HR-5 1261-1263
B-Star algorithm 1281

C

Channels 1317-1319
Channel voice messages 1388
Chess 1268-1270
Chess problem 1416-1417
Circular List program 1235-1236
Colmerauer, A 1272
Commodore 64
Cubic Splines program 1227
Drum Machine program 1239-1240
Go game 1406-1407, 1427
graphics facilities 1218-1219, 1256-1259
MIDI interface 1335-1337, 1343-1345, 1368-1371, 1412-1414
Numbers game 1283
Rotating Cube program 1258-1259, 1276-1279
simulation game 1212-1214, 1232-1233, 1244-1245, 1266-1267, 1294-1295, 1314-1316, 1338-1339, 1354-1356
Splitscreen program 1219
sound synthesis 1237-1240
Control commands 1304
Control Command Processor (CCP) 1324-1325, 1346-1347
CopyText program 1206
CP/M 1210, 1231, **1264-1265, 1284-1285, 1304-1305, 1324-1325, 1346-1347**
CP/NET 1347
Cray Blitz 1268-1270
Cray-1 1404-1405
Cubic splines 1226-1227
Cubic Splines program 1227
Cybernetics 1221-1222

D

Daisy-chain network 1386
Darwin, Charles 1322
Databases **1207-1208, 1224-1225**
DEC-10 PROLOG 1272-1273, 1327
Demons 1223
Digital Tracer program 1288, 1307
Digital Recorder program 1413-1414
Direct memory access 1218
Dr LOGO 1210-1211, **1434-1435**
Drum Machine program 1239-1240
Drum plotter 1308
Dynamic allocation 1234

E

Echo synthesiser **1341-1342**
Epson P40 1261-1263
Eurisko program 1222-1223
Evolutionary learning schemes **1348-1351**
Exclusive-OR gates 1252
Expert systems **1221-1223, 1301-1303**
Explanatory interface 1303
Eyes 1367

F

False eye 1367
Feigenbaum, Edward 1222
Fifth generation computers 1282, 1293
File Control Block (FCB) 1324-1325
File extensions 1304
File Management Utility 1373
FINDALL predicate 1352
Flat bed plotter 1308
Floyd 40 1261-1263
FORTH **1436-1437**

G

Galvanic Skin Response (GSR) 1202
Game tree 1282
GENE program 1348-1351
General Problem Solver (GPS) 1222
Go 1281-1282, **1366-1367**
Go game **1393-1395, 1406-1408, 1425-1428**
Greenblatt's MacHack program 1269
GSR systems 1202

H

Heuristic search 1221-1222
Hi-Res Print program 1334
Hisoft DEVPAC 3 1298
Hitch Hiker's Guide To The Galaxy **1400**
Hook codes **1398-1399, 1418-1420**

I

Illustrator 1374
Image understanding 1381
Impossible Mission **1220**
Inference engine 1302
Instruction counter 1365
Intelligence Ireland 1207
Intelligent robots 1421-1424
Interface 1 1398-1399
Interrupts 1378-1380

K

Kempston Interface 1262
Kildall, Gary 1264
Knowledge base 1301
Ko 1367
Kowalski, Bob 1272, 1293

L

The Last One 1374-1375, 1396-1397
Levy, David 1270
Liberty 1366
Lie detectors 1203
Life and death 1367
Linked structures 1234-1235
LISP **1376-1377, 1384-1385, 1415-1417**
Lists 1377, 1384
Local operators 1403
LOGO 1436-1437
Look-ahead 1282

M

Machine learning 1221-1223
Magic mouse 1410-1411
Maze Search program 1243
Measuring Stick program 1299
Microdrive channel structure 1419

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 61 TO 72

MicroPen package **1207-1208**
MICRO-PROLOG 1272-1273, 1327
Microtext 1374-1375
MIDI interface **1335-1337, 1343-1345, 1368-1371, 1386-1388, 1412-1414, 1432-1433**
Minder **1300**
Mindlink **1201-1203**
Minimaxing **1269-1270,** 1281
Mobile Robot program 1353
Mood-sensing software 1203
Moore, Charles 1436
MP/M 1347
Mycin program 1222-1223, 1302-1303

N

Natural language processing 1361-1364
Natural Language program 1364
Neural nets 1221
New World game 1214, 1232, 1245, 1267, 1295
full listing **1314-1316, 1338-1339, 1354-1356**
Noughts and crosses 1269
Numbers Game 1270-1271, 1283

O

Oberon Omni-reader **1429-1431**
Octuples 1403
Operating systems **1218-1219, 1237-1240, 1256-1259, 1276-1279, 1296-1299, 1317-1319, 1332-1334, 1357-1359, 1378-1380, 1398-1399, 1418-1420, 1438-1439**
Opto-isolator circuit 1336
Osborne 1 1264
Output device 1215
Overflow 1215

P

Packing 1215
Page 1215
Page printer 1215
Paper tape 1228
Parallel input/output 1228
Parallel processing 1228
Parameter 1252
Parity 1252
PASCAL **1204-1206, 1234-1236, 1246-1248,** 1252
Pass 1252
Password 1280
Patch 1280
Pattern Recogniser program 1402-1403
Pattern recognition 1280, 1381, **1401-1403**
Pearl, Judea 1281
Pen 1207-1208
Penform 1207
Perfect suite 1230-1231
Peripheral 1289
Peripheral interface adaptor 1289
Permutation 1289
Personal computer 1289
PET 1308
PIA 1289
Picture processing 1308, **1381-1383**
Picturesque Editor/Assembler 1298
Plasma display 1308
Platform games 1220
Plotter 1308
Ply 1282
Pointers 1234-1235
POKE 1340
Polish notation 1340
Polynomial 1340
Port 1340
Postfix notation 1340
Predicate logic 1272
Printed circuit 1340
Printer 1360
Probability 1360
Procedure 1360
Program 1365
Program counter 1365
Program development system 1365
Program generators **1374-1375, 1396-1397**
PROLOG **1272-1273, 1292-1293, 1312-1313, 1326-1327,** 1365
PROM 1365
PROM programmer 1365
Prompt 1392
Protocol 1392
Pulse 1392
Punched card 1392

Q

Query language 1409
Queue 1409
Quick data drive **1372-1373**
Quick operating system (QOS) 1372-1373
Quicksort 1409
The Quill 1374-1375

R

Radix 1409
RAM 1409
Random access 1440
Random number 1440
Raster scanning 1440
RM Nimbus **1328-1331**
Rosenblatt's Perceptron 1221-1223
Rotating Cube program 1258-1259, 1277-1279
Rule-breeding systems **1321-1323**

S

Samuel, Arthur 1269
Scout algorithm 1281
Searching 1241-1243, 1268-1271, 1281-1283, 1312-1313
Self-learning systems **1321-1323**
Self-reproducing automata 1424
Serial input/output 1228
Serial port 1438-1439
Shadowfire **1320**
Shannon, Claude 1268
SID chip 1237-1238
Sinclair Spectrum
Hi-Res Print program 1334
Go game 1408, 1428
memory map 1296
Numbers game 1283
operating system 1296-1299, 1317-1319, 1332-1334, 1357-1359, 1378-1380, 1398-1399, 1418-1420, 1438-1439
printers 1262-1263
robot arm sequence programmer 1216-1217
serial port 1438-1439
simulation game 1212-1214, 1232-1233, 1244-1245, 1266-1267, 1294-1295, 1314-1316, 1338-1339, 1354-1356
tape control 1357-1359
SOL System 1264
Speech recognition 1280
Star networks 1386
Streams 1332-1334
Suicide 1366-1367
Supercomputers **1404-1405**
Super suite 1390-1391
Sycero 1374-1375, 1396-1397
System messages 1388

T

Tandy 1000 1249-1251
Test functions 1385
TLO 1374-1375, 1396-1397
Transient commands **1284-1285**
Transient Program Area (TPA) 1346-1347
Transmission 1335-1337
Tree searching 1268-1271
TreeSorter program 1246-1247
Turing, Alan 1268

V

VIC-II chip 1218-1219
von Neumann, John 1268

W

Wheat Counsellor 1303
Wiener, Norbert 1221-1222
Wildcard characters 1305
Wirth, Niklaus 1246, 1252
WISARD program 1382, 1401-1403
WordStar 1265
Workshop digital tracer **1253-1255, 1274-1275, 1286-1288, 1306-1307**
Workshop MIDI interface **1335-1337, 1343-1345, 1368-1371, 1386-1388, 1412-1414, 1432-1433**
Workshop robot arm **1216-1217, 1226-1227**
Wren Executive **1229-1231,** 1265

Z

Zero page 1215
ZX printer 1261-1263